SHERWOOD DATA SYSTEMS LTD

BASIC INTERPRETER

VERSION C

Copyright 1984 by Sherwood Data Systems Ltd
Sherwood House, The Avenue
Farnham Common, Slough
England

CONTENTS

1. INTRODUCTION

2. INSTALLATION

2.1 Minimum Configuration
2.2 Theoretical Aspects Of Memory Expansion
2.3 Practical Ways To Add Memmory
2.4 User Programs In EPROM
2.5 Watchdog And Powerfail Circuitry
2.6 The DIP Switch Options
2.7 RS232 Connections
2.7.1 No Flow Control
2.7.2 Using Flow Control
2.7.3 Linking Two Archers Together

3. USING SDS BASIC

3.1 User Variables
3.2 Statements And Operators
3.3 Starting SDS Basic

4. SYSTEM LEVEL COMMANDS

5. PROGRAM STATEMENTS

6. FUNCTIONS

7. ERROR MESSAGES

8. TROUBLE-SHOOTING

1. INTRODUCTION

SDS Basic has been developed to enable users of the Archer single board computer to fully utilize the features of the Archer with the minimum amount of development time. It has various extensions to the Basic language, designed specifically for industrial applications. These are mainly connected with its ability to access an auxilary channel.

It has also been designed to enable the user program to be run from EPROM so that application software can be permanently built in, thus providing a very convenient way of producing a complete product. Naturally, in such applications, an autostart facility is needed so that the final user does not even need a console. This facility is provided, along with other supporting facilities such as a mechanism to output the program in a form suitable for downloading straight into a PROM programmer.

Basic is a language with many dialects. We have tried as far as possible to follow what we see as being fairly representative of modern versions of Basic. However, users of home computers will no doubt notice some differences between SDS Basic and what they are used to. Some of these differences are of course due to the addition of the extra facilities described above; some are due to memory limitations. We have squeezed as many facilities as possible into an 8k byte EPROM, but obviously have not been able to include all those which are commonly available on home computers. Such interpreters are sometimes 32k bytes long, much of which is concerned with colour graphics etc. which is not applicable in the kind of application for which SDS Basic is intended.

In writing the manual, we have assumed that the reader is familiar in general terms with the Basic language, which nowadays with the proliferation of home computers, we believe is a fair assumption. In the unlikely event that this is not the case, we refer the reader to the many excellent books on the subject which can be found in all good bookshops, with which we could not hope to compete. We have therefore concentrated in this manual on the topics which are peculiar to running this Basic on the Archer.

The installation and configuration of SDS Basic have been explained in great detail, but the standard commands, statements and functions have merely been defined so that the user can ensure that the syntax and other details are as expected. Where we have extended the language, we have naturally included more detail, adding where appropriate, application ideas and other hints on useage.

2. INSTALLATION

SDS Basic can be installed in a variety of ways with different amounts of user memory and with the option of using EPROM to hold the user program. Different size RAM chips, with or without battery back-up can be employed, and facilites exist for an autostart when power is applied. However, in order to get started, we commence by describing a minimum configuration.

2.1 MINIMUM CONGIGURATION

To use SDS Basic, the following hardware must be available:

- An SDS Archer, installed with mains connector etc.
- An SDS Basic Interpreter EPROM
- At least 1 memory chip, such as a 6116 or 2016 2k x 8 RAM.
- A display terminal or microcomputer emulating a terminal (standard Ascii, 8 bit, no parity, RS232 serial terminal).
- A suitable RS232 cable: see section on RS232 connections.
- An optional 8 way DIP switch.

Install the Basic EPROM in the lowest memory slot (IC2) which is the 28 pin socket nearest the battery. SDS Basic needs RAM to be installed at address 0F800H to 0FFFFH. Several possibilities exist to achieve this, the simpliest way being as follows, which relies on the fact that the top half of the memory address space will reflect down to the bottom half with the jumper links set as described. Install a 6116, or 2016 RAM in the top memory slot (IC5, furthest from the battery). Set the PCB links as follows:

LINK	POSITION	LINK	POSITION
1	2	14	1
2	2	16	out
3	2	17	out
4	2	18	5-6 and 7-8
5	2	19	any
7	1	20	any
8	1	21	any
10	1	22	any
11	1	23	2
13	1		

SDS Basic will work with a display terminal or personal computer running terminal emulation software, but when using the latter, difficulties can be encountered at high baud rates unless the emulation software is very fast or unless flow control is used. If possible, set your terminal or emulator to run at 9.6k baud (8 bits, no parity) and plug it into the left hand (J1) D-type connector of the Archer, using a suitable RS232 cable as defined

in the section on RS232 connections. If the optional DIP switches have been fitted to the Archer (next to the J1 D-type connector), then ensure that they are all off (open circuit).

If your terminal will not run at 9.6k baud, (which is likely if you are using a personal computer with terminal emulation software), then a lower baud rate must be selected or flow control must be used. Please refer to the sections on DIP switch options and RS232 connections respectively for details.

Switch the terminal and then the Archer on, and the Basic should sign on with the message:

```
SDS BASIC VERS. x.y
NEW? (Y/N)
```

where x.y is the version number. It is asking whether it should clear the RAM or not (User program and variables). For now, just type a "Y" which clears both program and data areas.

2.2 THEORETICAL ASPECTS OF MEMORY EXPANSION

In order to understand the more advanced configurations, the user is recommended to study the sections in the Archer manual on setting the various jumper links, and on the "reflections" which can be caused when using memory chips, smaller than the available address space. Some of the options can be quite complicated but they are very flexible.

This section discusses some of the ways to increase the user RAM to more realistic amounts. It is possible to have up to 24k bytes of RAM on the Archer board itself. By plugging a RAM extension board into the Archer, this total can be increased to 56k bytes.

SDS Basic expects to find its user program buffer at address 2000 (hex.), which is immediately after the interpreter itself. If it does not find either RAM or a valid user EPROM there, it hunts upwards through memory till it does. We will deal with the question of EPROM later; for now assume that we are only using RAM. SDS Basic assumes the use of either 2k or 8k byte RAM's. When it has located RAM in one of the Archer memory slots, it checks to see if it is an 8k or 2k byte chip, by checking for reflections. If it finds a 2k byte RAM it adds 1800 (hex.) to the starting address of the RAM. In other words it uses the top reflection, to make the slot contiguous with the next slot, so that if two RAM's are used, the address space will not have gaps.

The SDS Basic variables are stored at the top of memory, which in the absence of a RAM extension card reflects down into the top Archer memory slot.

The nett result of all this theory, is that many options exist to

increase the amount of RAM available. It is sufficient to plug in the extra chips, and set the appropriate links for those chips (see the Archer manual). However, for correct operation, the user should ensure that the address space is contiguous. The only combination of chips where this is not possible is with three 2k chips. Basic will still work with three 2k chips, but there will be a gap in the memory. If the user program and work space exceed 4k they will not be able to use any of the third chip. Similarly if the variables exceed 2k they will not be able to use any of the other two chips, and the interpreter will probably crash.

When Basic is running, the command "FREE" prints three numbers. The first, in decimal is the amount of unused memory, assuming that there are no gaps in the addressing, and that a RAM extension card is being used. In the absence of a RAM extension card, 32k (32768) should be subtracted from this number. The other two numbers are the end of the user program and the start of the variable area, both in hexadecimal. By examining these numbers the user can tell whether address gaps are likey to be a problem. Note that the middle number which represents the end of the user program grows upwards as the program grows, and the third number grows downwards as the number of symbols increases. Both numbers represent the actual memory in use at the time. Running the program can add variables and thus increase the demand for memory. Similarly an allowance should be made for the run-time work space which is located immediately after the progam area. (Unfortunately it is impossible to give any guidelines as to this memory requirement since it is heavily dependent on the amount of nesting in the program.)

2.3 PRACTICAL WAYS TO ADD MEMORY

Whether or not you have studied the theory in the previous section, adding more memory is very simple in practice. In this section we list some typical choices. To add memory, plug in the extra chips as in the following table, and set the jumper links according to the type of chips as described in the Archer manual.

TOTAL MEMORY REQUIRED	SLOT 2 (IC 3)	SLOT 3 (IC 4)	SLOT 4 (IC 5)
2k	-	-	2k
4k	-	2k	2k
8k	-	-	8k
10k	-	8k	2k
12k	2k	8k	2k
16k	-	8k	8k
18k	2k	8k	8k
24k	8k	8k	8k

Other combinations are possible, but they should only be attempted when the theory explained in the previous section has been understood. As noted previously, the use of three 2k byte RAM chips to obtain 6k bytes should normally be avoided.

To use a RAM/ROM extension card, set the links for bank 1, and put RAM in at least the top position. Bank 1 will be automatically enabled when SDS Basic starts running. Extra chips should be added from the 'top downwards', the best choice of RAM being the 8k byte RAM's. With four of these in the RAM/ROM extension card, and three in the Archer itself, upto 56k bytes of RAM can be obtained.

2.4 USER PROGRAMS IN EPROM

When developed, a program can be transferred to EPROM, and reinstalled in the Archer. The user program is first output to a PROM programmer using the "DUMP" command. (See the section on DIP switch options for baud rate selection.) It can then be programmed into one or more EPROM's. When using a single EPROM, any suitable 2k, 4k or 8k byte device may be used (eg. 2716, 2732 or 2764). When the program requires more than one device, 8k byte EPROM's (eg 2764) must be used in all but the last position to ensure contiguous addressing. As many EPROM's as desired may be used to hold the program, as long as some RAM is available as a workspace, either in the top slot or using a RAM extension card.

For larger programs, the slot spacing can be altered to 16k bytes (jumper link 18) to permit the use of two 27128 EPROM's in slots 2 and 3. A 2k or 8k byte RAM can then be used in the top slot.

After installing EPROM do not forget to disable the battery backup if it was previously being used on that slot. When the Archer is turned on it will recognise the existance of EPROM and move its workspace up till it finds RAM (instead of placing it immediately after the user program). Any unused EPROM between the end of the user program (determinable with the "FREE" command), and the start of the RAM can be used to hold machine code subroutines which can be accessed with the "CALL" command.

SDS Basic recognises the existance of a valid EPROM by the code 0AA (hex.) in the first location. If this is absent, the slot will be ignored. This fact can be used to incorporate machine code subroutines into both RAM and EPROM systems, or to hold other programs such as loaders or other utilities.

When the Archer is turned on, it will sign on as usual with the message:

```
SDS BASIC VERS. x.y
NEW? (Y/N)
```

This time the reply is normally "N". Note that after a "No" reply, the variables will not be initialised. They can be cleared with the "CLEAR" command, or if battery backup is in use, left for further processing by the program in EPROM.

If the user replies "Y" to the "NEW? (Y/N)" question, SDS Basic will "delete the EPROM" by moving on till it finds RAM. The program in EPROM can be recovered with the "RESET" command.

2.5 WATCHDOG AND POWERFAIL CIRCUITRY

When you have become familiar with SDS Basic, links 16 and 17 can be inserted to activate the powerfail and watchdog circuits respectively. Links 7, 10 and 13 should also be moved to position 2 to turn on the battery backup facility if RAM is being used in the associated slot.

SDS Basic regularly triggers the watchdog timer and prevents it resetting the processor. Once control is transferred to a user subroutine with the "CALL" command, this triggering ceases, and it is up to the user subroutine to take over.

When the powerfail circuitry is enabled, the software will continue where it left off after a power failure. (See section on DIP switch options.) SDS Basic sets the baud rates to the values specified by the DIP switches before continuing.

If the powerfail circuitry is active and the battery backup is enabled, switching the power off and on again will not reset the Basic. If a crash has occured for any reason or if Basic appears to be completely dead, then remove the powerfail link, and if necessary the battery backup links before trying to restart it.

2.6 THE DIP SWITCH OPTIONS

It is possible to add two optional 8 way DIP switches in positions near the D-type connector J1. They are not installed in the standard ex-factory configuration since some users require straight DIP switches and others prefer right angled versions which in a Cased Archer, can be made to be externally accessible.

When the DIP switch SW1 is not installed, SDS Basic powers up with both serial channels set to 9.6k baud, and does a normal start (no autostart or continue). If these default parameters are

not wanted, then DIP switch SW1, nearest connector J1, must be installed, alternatively, wire links can be soldered across selected positions. (Sherwood Data Systems Ltd. can deliver the Archer with either style of DIP switch already fitted on request.) With SW1 fitted, the options available are as follows:

DIP SW1 viewed from the edge of board with the serial connectors

left								right
		p	q	r	p	q	r	
auto	cont.	console baud rate			auxilary baud rate			

auto: open = normal cold start.
closed = automatically start program in EPROM or battery backed RAM.

cont: open = normal or autostart (selected above).
closed = continue previous program if possible.

Notes: For the "continue" option to work, battery backup must be enabled on at least the top memory slot. Obviously it may also be needed on other slots if they contain program or data in RAM. Link 17 must be installed, and the system must be protected from mains interference with suitable filters and/or shielding. When enabled, this option will override the "auto/normal" option. If for any reason, Basic cannot continue where it left off (eg. the checksum failed), then the "auto/normal" switch will be examined to determine a second choice start up condition.

The baud rates for the two serial channels are selected as follows:

p	q	r	
closed	closed	closed	110 baud
closed	closed	open	200 baud
closed	open	closed	300 baud
closed	open	open	600 baud
open	closed	closed	1200 baud
open	closed	open	2400 baud
open	open	closed	4800 baud
open	open	open	9600 baud

The DIP switches are only read when the power is applied or when a "RESET" command is issued. Changing the DIP switches has no effect once Basic is running unless the "RESET" command is given. The "RESET" command can be used to change the baud rates. The autostart will work if enabled, but not the "continue" option.

2.7 RS232 CONNECTIONS

To enable SDS Basic to work with a variety of terminals, personnal computers and other serial devices, version C.1 onwards incorporates flow control on both serial interfaces. This enables higher speeds to be used than would otherwise be possible. SDS Basic uses the following RS232 signals:

pin	RS232 NAME	IN/OUT	USE
1	FG	-	protective ground (chassis)
2	TxD	in	data transmitted from terminal
3	RxD	out	data received by terminal
4	RTS	in	enables Archer receiver
5	CTS	out	controls input data flow
7	SG	-	signal ground
8	RSLD	out	SDS Basic "up and running"
20	DTR	in	controls output data flow

The theory and practical details of these signals are described in more detail in the Archer User's Manual. For our purposes we define three possible connections. In each case connect pins 2,3 and 7 to pins 2,3 and 7 of the other device respectively, if it is a terminal (DTE), or swap pins 2 and 3 if it is a DCE (modem or computer-like device).

2.7.1 No Flow Control

If the device connected to the Archer can receive data at the chosen baud rate without requiring pauses and if it only sends data to the Archer slowly, then no flow control is necessary. (A true display terminal is likely to meet this requirement.)

Connect pin 8 (or 6 which is identical) to pins 4 and 20. This permanently enables the Archer receiver and transmitter circuits.

2.7.2 Using Flow Control

Most devices, including the Archer running SDS Basic, cannot accept data at high baud rates indefinitely. SDS Basic will accept data at upto 9.6k baud until a carriage return is received. It then requires time to process the line or to store it in the correct sequence in memory (interrupts are not used). Whilst it is thus busy, it outputs a low signal on pin 5 (CTS) of the serial connector. This can be used to cause the data source

to suspend transmission. Conversely, a low signal input on pin 20 (DTR), will cause SDS Basic to pause when outputting data.

These two signals can be used to synchronise the Archer to the other equipment. How they are connected will depend on the exact details of the other equipment, but if the Archer is connected to a normal terminal style device (DTE), then pin 5 will go to pin 5 and 20 to 20.

Pin 4 can be looped back to pin 8 as in the no flow control connection, or it can be used to provide link control in more advanced applications.

2.7.3 Linking Two Archers Together

When linking two Archers together, with both running SDS Basic, the following cable should be used:

Archer One		Archer Two
1	----------	1
2	----------	3
3	----------	2
4	----------	8
5	----------	20
7	----------	7
8	----------	4
20	----------	5

Alternatively, pins 4 and 8 can be linked locally to save two wires, and the link between pins 1 and 1 can be omitted.

3. USING SDS BASIC

3.1 CONVENTIONS AND SYNTAX

The following abbreviations are used throughout this manual:

C/R	- The carriage return or new-line key	(ASCII ODH)
L/F	- The line feed or cursor down key	(ASCII OAH)
B/S	- The backspace or cursor left key	(ASCII O8H)
DEL	- The delete or rub-out key	(ASCII 7FH)

Control characters, which are obtained by simultaneously pressing the CONTROL key and the relevant character key, are represented in the manual and echoed by SDS Basic as "^" or up-arrow, followed by the character. L/F is ignored by the interpreter. SDS Basic will accept both upper and lower case letters, which are treated as being equivalent.

If typing errors are made when using SDS Basic, they can be corrected by pressing either B/S or DEL. Both are equivalent and assume that the terminal can backspace to physically remove the erroneous character from the screen. When using a hard copy terminal, the erroneous character will of course not appear to have been deleted, but will in fact have disappeared from the RAM. Entering a line number alone will delete that line. Lines may be entered out of sequence and will be sorted on entry.

There are several other special control characters:

^C	cancel command or interrupt running program
^H	same as B/S
^J	same as L/F
^M	same as C/R
^O	halt/resume program output
^S	suspend program execution (any character resumes)
^U	delete line being typed
^X	delete line being typed
^Z	end of file marker

The ^O can be used to suppress the output (on either the console or auxilary channel) of unwanted output. This can be used for example to speed up processing, particularly when a slow printer is attached to the auxilary port. A second ^O reenables output. Note that if the ^O is not cancelled with a second ^O, various types of output will be supressed even at system level, for example the LIST command will print a blank program. Reenable with ^O to regain normal operation.

The ^S places the system into a hold condition. Use it to suspend processing while you answer the telephone or change paper in the printer. Any character will restart processing.

3.2 USER VARIABLES

User variables consist of a letter (upper or lower case) followed optionally by a digit 0-9. String variables must also have a dollar sign appended.

Examples: A , b1 , q , T7 (Numeric)
A# , b1# , q# , T7# (String)

Numeric arrays are supported with up to two dimensions, but must be declared with a "DIM" statement before use. Such a declaration is not required for single dimensional arrays with subscripts in the range 0 to 10 incl. String arrays are not allowed.

Examples: A(0) , a1(7) , q(3) , Z7(23)

Statement numbers must lie in the range 1 to 9999. It is recommended that statements are spaced at intervals of 10 to allow insertions. Several statements may be placed on one line if separated by colons.

Strings may be up to 235 characters long. String constants must be enclosed in either single or double quotes as follows:

"This is a string constant"
or 'This is a string constant'

The same delimiter must be used at each end. This permits the other punctuation mark to be used within the string as follows:

"This is Sherwood Data System's Basic"
or 'He said "Switch on the Archer"'

3.3 STATEMENTS AND OPERATORS

All commands may be abbreviated to four letters or more if desired. If extra letters are present, they must be correct.

Example: PRINT = PRIN
RETURN = RETU = RETUR

The following operators are allowed: +,-,*,/,^,(and).

A plus sign is used to concatenate strings.

3.4 STARTING SDS BASIC

Once SDS Basic has been correctly installed, with sufficient memory for the intended program and with all the Archer jumper links in their correct positions, most users should have no problems in running it. To ensure that this is the case, we provide in this section a few hints and some extra information on the start up conditions of SDS Basic.

It is up to the user to ensure that the memory is initialised (both program and data areas), before the program is run. Answering "Y" to the sign-on question will ensure this, by clearing both, but this is not always wanted. The following table shows the effect of various actions on the program and data areas:

ACTION	PROGRAM	DATA
Sign-on NEW? (Y/N)...Y	cleared	cleared
Sign-on NEW? (Y/N)...N	-	-
Autostart	-	-
Statement "CLEAR"	-	cleared
Command "NEW"	cleared	cleared
Command "DEL"	cleared	-
Command "RUN"	-	cleared
Command "EXEQ"	-	-
Power failure (no backup)	destroyed	destroyed
EPROM	preserved	-
battery backup	preserved	preserved

(a dash means no effect)

The user must ensure that both areas are either cleared or contain valid information before use. This can be achieved by a combination of commands or by ensuring that the information is preserved in battery backed RAM or in EPROM.

For example replying "N" to the sign on message will leave the memory uninitialised unless the battery backup is enabled or EPROM is being used. In the latter case the data area still remains uninitialised. Use CLEAR at the beginning of the program or start the program with RUN not EXEQ. Similarly, autostart must not be used without a CLEAR at the beginning of the program unless the RAM has battery backup for the variables.

4.0 SYSTEM LEVEL COMMANDS

This section describes the commands which are available to control and monitor the execution of the user's program. In addition, all program statement type commands may be issued at system level for immediate execution.

Command CONT

Example: CONT

This command restarts a program where it left off when stopped either by a STOP command or through a control/C. It can also be used to continue a program which stopped at an error which has subsequently been corrected.

Command DEL

Example: DEL

DEL deletes the user program without effecting the variables. It can be used to chain programs together and to pass the results of one program to the next. If a user program has been installed in EPROM, this too will appear to be deleted by DEL. One application of the DEL command is to run a commonly used program from EPROM, then to delete this program, and to type in or load another in order to continue processing the previous variables.

Command DUMP

Example: DUMP

The DUMP command dumps the entire user program in industry standard hex. format via the auxilary channel. Variables are not included in this DUMP. DUMP is used to transfer a program to a PROM programmer so that the user program can be cast in silicon for permanent or at least semi-permanent use.

Command EXEQ

Example: EXEQ

EXEQ starts a program without initialising the variables. It can be used instead of the more usual RUN command to chain programs together or to manually restart processing after a power failure.

Command FREE

Example: FREE

FREE prints three numbers on the console. The first is the amount of unused RAM available, assuming a fully populated 64k byte memory. The other two are in hexadecimal and represent the upper limit of the user program and the lower limit of the user variables respectively. For detailed information on how to interpret these numbers when the RAM is not fully populated, please refer to the section on the theoretical aspects of memory expansion.

Command LIST

```
Example: LIST
or       LIST xxxx
or       LIST xxxx,yyyy
```

When LIST is issued without an argument, it prints the entire user program on the console. When one argument xxxx is given then only this line is printed. With two arguments, the portion of the program bounded by the two arguments is printed.

Command LLIST

```
Example: LLIST
or       LLIST xxxx
or       LLIST xxxx,yyyy
```

This is identical to the LIST command except that the output is sent to the auxilary channel. It is assummed that a line printer is attached to this auxilary channel.

Command LOAD

Example: LOAD

The LOAD command is used to load a program from another computer or storage device into the Archer memory via the auxilary channel. C/R is used as a start-of-text-character to prevent spurious characters, which may be produced by devices such as cassette recorder interfaces, from interfering with the load operation. The program, if prepared on another computer should therefore start with a blank line. Control/Z is used as an end-of-file marker to transfer control back to the console. If this is absent, control can be regained by typing control/C. Note that any previous program should have been deleted with the NEW or DEL command unless both programs are to be merged together.

When using the LOAD command it is sometimes useful to be able to monitor the information being loaded. This can be done with the ECHO statement, which can turn on the echo to either the console or auxilary channels. Do not echo to the console however, when the console baud rate is lower than the auxilary baud rate, since the console will not be able to keep up and characters will be lost.

Command NEW

Example: NEW

The command NEW deletes both program and data areas in preparation for a new program. If the user program is in EPROM it can still be apparently deleted with the new command (and recovered with the RESET command).

Command RESET

Example: RESET

This command restarts the Basic interpreter. The DIP switches, if present are read and the baud rates set. The positions of RAM and user EPROM are redetermined and either a normal or autostart is performed. The user program and variables are not lost unless the user answers "Y" to the initial "NEW? (Y/N)" question or unless EPROM is activated by a "N" reply, which deletes the RAM program.

Command RUN

Example: RUN

This is the normal way of starting a program. The variable data area is cleared by this command before the program starts.

Command SAVE

Example: SAVE
or SAVE xxxx,yyyy

SAVE outputs the entire user program or a part thereof via the auxilary channel in a format suitable for reloading with the LOAD command. (In contrast to the LLIST command.) It is also compatible with most personal computers.

5.0 PROGRAM STATEMENTS

In this section, the statements which are normally used as commands within a program are described. They can also be used as system level commands like those in the previous section.

Statement AINPUT

Example: 120 AINPUT x,a#

This statement is equivalent to the usual INPUT statement, except that it operates on the auxilary channel instead of the console. Unlike the INPUT command, echoing is not automatic, but is controlled by the ECHO statement, the default being no echo. Please refer to the INPUT command for examples of syntax and other details.

Statement AINPUT LINE

Example: AINPUT LINE a#

This statement is equivalent to the usual INPUT LINE statement, except that it operates on the auxilary channel instead of the console. Unlike the INPUT LINE command, echoing is not automatic, but is controlled by the ECHO statement, the default being no echo. Please refer to the INPUT LINE command for examples of syntax and other details.

Statement CALL

Example: 120 CALL xxxx

The CALL statement calls a machine code subroutine at location xxxx. It is up to the user to ensure that the machine code subroutine does not clash with the Basic program. One way to do this is to put the machine code into EPROM as described in the section on user programs in EPROM.

Statement CLEAR

Example: 10 CLEAR

The statement CLEAR clears all user symbols and regains the space for further use. It is intended for use at the beginning of a program which normally operates under autostart, since autostart

does not clear variables. Since program flow control data is stored in the same area as the user symbols, CLEAR must only be used at the top level in a program (never in a FOR....NEXT loop or subroutine etc). Is is recommended that it only be used in the first line of a program.

Statement DATA

Example: 120 DATA xxx,yyy,zzz

This defines one or more constants (xxx,yyy,zzz), which can subsequently be read with the READ command.

Statement DEF FN

```
Example: 120 DEF FNQ(X) = SQR(X^3)
         130 LET Z= FNQ(4)
```

Up to 26 functions can be defined with this statement. In the example the function Q is defined and used later on line 130. The variable Z would evaluate to 8 in this case.

Statement DIM

```
Example: 120 DIM A(nn)
or       120 DIM A(nn,mm)
```

The DIM statement defines a one or two dimensional array with nn or nn by mm elements. For one dimensional arrays with subscripts from 0 to 10, no DIM statement is needed. The maximum number of elements is limited to 255, but should be restricted to lower values if possible, to conserve memory. String arrays are not supported.

Statement ECHO

Example: ECHO(x)

ECHO controls the echoing of input on the auxilary channel for the AINPUT and AINPUT LINE statements, which unlike their corresponding INPUT and INPUT LINE statements do not normally echo information back. The reason for this is that whilst a person who puts information in via the console normally likes to see what he has typed, an instrument which automatically inputs information via the auxilary channel is not interested in such visual feedback. In fact the output part of the auxilary channel may be connected to an entirely different device such as a

printer. The ECHO statement is therefore provided to enable such visual feedback for those applications which require it. It also works with the LOAD command. The argument x in the ECHO statement, which should have a value between 0 and 3 is interpreted as follows:

x	type of echo
0	no echo.
1	echo to console.
2	echo to auxilary channel.
3	echo to both console and auxilary channel.

Note that error messages and the question mark prompt are also controlled by the ECHO statement. The system is reset to the no echo condition each time the user program is started, so the ECHO statement should be incorporated in the user program rather than executed at system level.

Statement END

Example: 120 END

END defines the physical end of the user program. It is not normally needed, though if absent, no control/Z end-of-file marker will be produced by the SAVE command. If present, it must be the last statement in the program.

Statement FOR

```
Example: 120 FOR I= xx TO yy <STEP zz>
         130 PRINT I
         140 NEXT I
```

FOR defines the beginning of a loop, which is terminated by a NEXT statement. A variable, in the example I, is first set to xx, and then increased by an amount zz each time the loop is traversed. The loop terminates when the value yy is reached. The increment zz, defined by the optional STEP zz is asummed to be 1 when the STEP zz part of the statement is not included. FOR statements may be nested up to eight levels deep.

Statement GOSUB

```
Example: 120 GOSUB xxx
         ....
         ....
         xxx PRINT "SDS Basic"
         yyy RETURN
```

The GOSUB statement calls a subroutine at line xxx. When the subroutine is complete, control is transferred back with a RETURN statement.

Statement GOTO

Example: 120 GOTO xxx

GOTO does an unconditional jump to the line xxx.

Statement IF

Example: 120 IF xxx = yyy THEN zzz
or: 120 IF xxx = yyy THEN Statement

The test equation between the IF and the THEN statements is evaluated, and if found to be true, then the statement after the THEN is executed. When a number zzz appears after the THEN instead of a statement, then this is understood to mean GOTO zzz.

If the equation is not true, then control is passed to the next line. The test statement may contain the following boolean operators:

<>	not equal
<	less than
>	greater than
>=	greater than or equal
<=	less than or equal

Statement INPUT

Example: 120 INPUT x,"SDS Basic",y,z,a$,b$

This statement prompts at the console with a question mark to ask for the value of one or more variables x, y, z etc. The variables can be either numeric or string variables. When they are input, C/R is used as a final terminator and either C/R or comma as separators between variables. If string variables containing commas are to be input, the variables must be enclosed in quotation marks.

A suitable response for the example above might be:

```
? 1234            C/R
? 3.14159         C/R
SDS Basic? 30     C/R
? Temperature     C/R
? "abc,def"       C/R
```

Statement INPUT LINE

Example: 120 INPUT LINE x$

INPUT LINE differs from INPUT, in that is reads a whole line in one go as a string variable. C/R is used as a terminator and the line may contain any other character except ^C, ^O, L/F, NULL, B/S, DEL, ^S, ^U, ^X and ^Z.

Statement LET

Example: 120 LET x = yy
or 120 x = yy

This defines the variable x to be equal to yy. The keyword LET may be omitted if desired as in the second example.

Statement LPRINT

Example: 120 LPRINT xxx
other examples as in PRINT

This statement is identical to the PRINT statement, except that the output is directed to the auxilary channel. This can be attached to a line printer, or to some other instrument which also uses the input side of the auxilary channel.

Statement NEXT

Example: 120 NEXT I

NEXT defines the end of a FOR...NEXT loop. Please refer to the FOR statement for details.

Statement ON

Example: 120 ON qqq GOTO xxxx,yyyy,zzzz
or 120 ON qqq GOSUB xxxx,yyyy,zzzz

The expression qqq is evaluated and the result used as a pointer to one of the line numbers xxxx, yyyy or zzzz etc. This line is then used as the appropriate argument for the GOTO or GOSUB statement.

For example, if qqq evaluates to 3, then the third line number zzzz is chosen and will be used in the GOTO or GOSUB statement.

Statement OUT

Example: 120 OUT xx,yy

OUT is used for direct access to the output ports of the Archer. It output the value yy to the port with device address xx.

Statement POKE

Example: 120 POKE xxxx,yy

POKE xxxx,yy stores the value yy at the Address xxxx (Decimal) in the RAM. It is up to the user to ensure that the normal operation of the interpreter is not disturbed by this action.

Statement PRINT

```
Example: 120 PRINT xxx,yyy,zzz
or       120 PRINT xxx;yyy;zzz
or       120 PRINT "tttttt"
or       120 ? xxx,yyy,zzz
```

In the first two examples, one or more expressions xxx, yyy, zzz are evaluated then output to the console. If the expressions are separated by commas, then the individual results are spaced at 14 tab-stop intervals. If a semi-colon is used as a separator, then the individual results are spaced as close as possible. The third example shows the output of a string constant.

At the end of each PRINT statement, a new line is automatically produced unless the PRINT statement is terminated by a comma or semi-colon. The final example shows the use of a question mark as an abbreviation for the PRINT.

Statement RANDOMIZE

Example: 120 RANDOMIZE

This reseeds the random number generator to give it a new starting value. Please refer to the RND(x) function.

Statement READ

Example: 120 READ x,y,z

The variables x,y,z are loaded with values from the DATA Statement. For example, with DATA 123,456,789, x would become

123, y 456 and z would become 789.

Statement REM

Example: 120 REM This is a comment

Everything to the right of the REM statement is ignored, enabling the addition of comments in the program.

Statement RESTORE

Example: 120 RESTORE

RESTORE works with the READ and DATA statements. It resets the internal pointer so that the next READ gets its data from the first expression in the first DATA statement.

Statement RETURN

Example: 120 RETURN

RETURN signifies the end of a subroutine. It returns control to the next statement after the GOSUB.

Statement STOP

Example: 120 STOP

STOP defines the logical end of a program. It only needs to be used when this occurs in the middle of the program. One might for example put the subroutines physically at the end of the program and use the STOP at the end of the main program.

6.0 FUNCTIONS

This section describes the functions that may be used in expressions within statements. The all return one number which is the value of the argument. They require between zero and three arguments, but in the case of zero arguments, a dummy argument is required (exception RND). Trigonometric functions assume the use of radians.

Function ABS

Example: 120 LET y = ABS(x)

Result: Returns the absolute value of x.
ie. When x is positive, ABS(x) = x
When x is negative, ABS(x) = -x
When x is zero, ABS(x) = 0

Function AINKEY$

Example: 120 LET y$ = AINKEY$(x)

In this function, x is a dummy argument. AINKEY$ is similar to the INKEY function, except that it reads data from the auxilary channel. If a character has been input on the auxilary channel, then AINKEY$ returns a string containing that character. The character is not echoed. If no character has been input, then a null string is returned. Unlike the INKEY$ function, AINKEY$ will accept all ASCII characters including all control characters and ASCII NULL. It can therefore be used to read binary data. The most significant bit is however reset.

Function ASC

Example: 120 LET y = ASC(x$)

Result: Returns the numerical ASCII-code of a string variable. For example, ASC("A") = 65. This function is the inverse of CHR.

Function ATN

Example: 120 LET y = ATN(x)

Result: Forms the arctangent of x. (y is in radians).

Function CHR$

Example: 120 LET x$ = CHR$(y)

Result: Changes a numerical value into an equivalent ASCII string x$. This function is the inverse of ASC(x).

Function COS

Example: 120 LET x = COS(y)

Result: Produces the cosine of y. (y is in radians).

Function EXP

Example: 120 LET x = EXP(y)

Result: Forms the exponential e^y

Function FNQ(x)

Example: LET x = FNQ(x)

This is a user defined function. The letter Q may be replaced by any of the 26 letters A to Z, to give up to 26 different functions. See the DEF FNQ(x) statement for an example.

Function INKEY$

Example: 120 LET y$ = INKEY$(x)

In this function, x is a dummy argument. If a character has been input on the console, then INKEY$ returns a string containing that character. The character is not echoed. If no character has been input, then a null string is returned. INKEY$ will accept all ASCII characters including all control characters except ^C, ^O and ^S, which have their usual meanings. The most significant bit is always reset.

Function INP

Example: 120 x = INP(y)

Result: This function, the complement of the OUT statement, returns a value read from the input port addressed by y.

Function INSTR

Example: 120 LET x = INSTR(y,z1$,z2$)

Result: INSTR searches for the string z2$ in the string z1$, starting at the position y. It returns a value representing the position of z2$ in z1$. When z2$ does not occur in z1$ the value zero is returned.

Function INT

Example: 120 LET x = INT(y)

Result: This produces the largest integer, smaller than or equal to y.

Function LEFT$

Example: 120 LET x$ = LEFT$(y$,n)

Result: LEFT$ extracts the first n characters from the string y$.

Function LEN

Example: 120 x = LEN(y$)

Result: The value of this function is equal to the length of the string y$.

Function LN

Example: 120 LET x = LN(y)

Result: This results in the natural logarithm (base e) of y.

Function LOG

Example: 120 x = LOG(y)

Result: This function results in the logarithm to the base 10 of the argument y.

Function LPOS

Example: 120 x = LPOS(y)

Result: LPOS returns the head position of the printer attached to the auxilary channel. The left most position has the value zero. The function is only valid after the program has executed its first carriage return, since the position of the head is unknown at switch on. In the case of true line-printers which only print when they have received a complete line, the function represents the logical position of the head at that time rather than its physical position. y is a dummy argument and has no meaning.

Function MID$

Example: 120 LET x$ = MID$(y$,m,n)

Result: This extracts n characters from the string y$, starting at position m.

Function PEEK

Example: 120 LET x = PEEK(y)

Result: the value of this function equals the contents of the memory location addressed by y.

Function PI

Example: 120 LET x = PI

Result: PI is a constant with the value 3.14159

Function POS

Example: 120 LET x = POS(y)

Result: POS produces a number representing the cursor position on the console. The left most position has the value zero. y is a dummy argument and has no meaning.

Function RIGHT$

Example: 120 LET x = RIGHT$(y$,n)

Result: This function extracts the right part of the string y$ starting at the n'th character.

Function RND

Example: 120 LET x = RND(y)
or 120 LET x = RND

Result: RND produces a pseudo-random number between 0 und y. If the argument (y) is absent, as in the second example, a value between 0 and 1 is returned.

Function SGN

Example: 120 LET x = SGN(y)

Result: This results in -1 , 0 or +1 depending on whether y is negative, zero or positive.

Function SIN

Example: 120 LET x = SIN(y)

Result: The result is the sine of y. (y is in radians).

Function SPACE$

Example: 120 LET x$ = SPACE$(x)

Result: This produces a string of x spaces.

Function SQR

Example: 120 LET x = SQR(y)

Result: SQR extracts the square root of y.

Function STR$

Example: 120 LET x$ = STR$(y)

Result: This converts a number into a corresponding string. It is the converse of VAL.

Function STRING$

Example: 120 LET x$ = STRING$(y,z)

Result: STRING$ produces a string of length y with all the characters having the ASCII value z.

Function TAB

Example: 120 PRINT TAB(x)
or 120 LPRINT TAB(x)

Result: This function, when used inside a PRINT or LPRINT statement, moves the cursor or printer head forwards to the position x. If the cursor or head is already to the right of position x, TAB does a carriage return / line feed first.

Function TAN

Example: 120 LET x = TAN(y)

Result: TAN calculates the tangent of y. (y is in radians.)

Function VAL

Example: 120 LET x = VAL(y$)

Result: This returns the numerical value of a "numerical" string. For example, VAL("1234.56") produces 1234.56

7.0 ERROR MESSAGES

When errors occur within a running program, an error message of the form:

xxxx ERROR IN LINE yyyy

is produced, where xxxx is one of the descriptions below. No line number yyyy is issued if the statement was given at system level.

ARGUMENT - A function has too many or too few arguments.
- The memory does not contain a valid program. (most likely caused by answering "N" to the question NEW? (Y/N) at start-up without battery backup or EPROM.

DATA - A READ statement has run out of data in a DATA statement.

DIGIT - A digit 0-9 was expected at that point, but some other character was found.

LINE NUMBER - A non-existant line number was specified in a GOTO, GOSUB or IF...THEN statement.

MATHS - Division by zero.
- Taking the square root of a negative number.
- Taking the LOG or LN of a negative number.

NEXT - A NEXT statement without a corresponding FOR statement was executed.
- FOR...NEXT loops are nested more than eight levels deep.

OVERFLOW - A number greater than 5.76461E18 was produced. Program execution continues.

RETURN - A RETURN statement without a corresponding GOSUB command was executed.

SYNTAX - The interpreter cannot understand the statement.

UNDERFLOW - A number smaller than 2.71050E-20 but not equal to zero was produced. Program execution continues.

8. TROUBLE-SHOOTING

Whilst hardware malfunctions and program bugs can occur, most problems are caused by misunderstandings. This section of the manual is designed to pinpoint likely problems.

Firstly, the obvious - if the system is completely dead, check:

Is the Basic EPROM oriented correctly in position IC2 ?
Is at least one RAM chip installed (IC5) ?
Are the jumper links installed correctly on the Archer ?
Is the console baud rate correct ?
Are the DIP switches (if installed) set correctly ?
Are the fuses, mains lead, RS232 lead etc. ok ?
Are pins 4 and 20 of the RS232 interfaces active (high) ?

If the system is alive but not behaving coherently, check:

Is the RAM initialised (answer "Y" to sign on message) ?
Is the console baud rate correct ?

If the system periodically crashes, check:

Is the RAM large enough for the program and data ?
Are there gaps in the RAM addressing ?
Is there a CLEAR statement other than at the top level ?
Are you using CALL without refreshing the watchdog ?
Are the RAM / EPROM chips fast enough for 4 Mhz. operation ?

If the battery backup does not work, check:

Is the battery charged (about 3.5 volts) ?
Are the relevant jumper links installed (7, 10 and/or 13) ?
Are irrelevant jumper links removed ? (Do not backup EPROM.)

If the LOAD command will not load properly, check:

Is the auxillary baud rate too high ?
Is aux. channel ECHO enabled to a lower baud rate console ?
Does the program start with a blank line ?

If the SAVE command will not work properly, check:

Is the auxilary baud rate too high or flow control needed ?
Does your other computer need some special EOF marker ?

If the printer or personal computer produces garbled text, check:

Is the auxilary or console baud rate too high ?

If the output wholly or partially disappears, check:

Is the ^O output suppress in operation ?
Is the ^S program suspend in operation ?